Static Program Analysis

Lecture 19: Pointer & Shape Analysis I

Summer Semester 2018

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-18/spa/

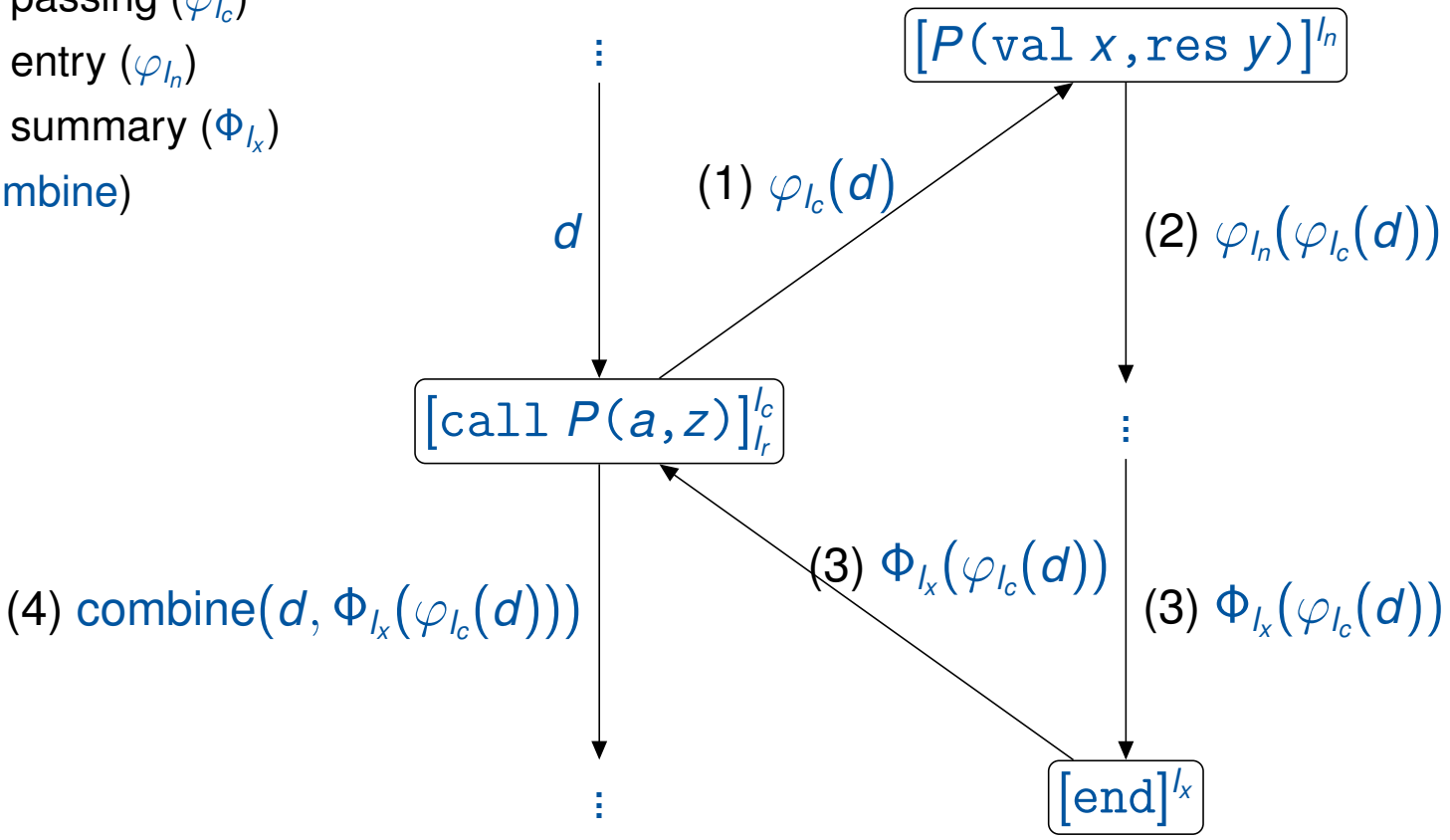
Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

Recap: Fixpoint Solution to Interprocedural Dataflow Analysis

Information Flow

1. Parameter passing (φ_{l_c})
2. Procedure entry (φ_{l_n})
3. Procedure summary (Φ_{l_x})
4. Return (combine)



Recap: Fixpoint Solution to Interprocedural Dataflow Analysis

Formal Definition of Equation System

Dataflow equations (for each $l \in Lab$):

$$\mathsf{AI}_l = \begin{cases} \iota & \text{if } l \in E \\ \mathsf{AI}_{l_c} & \text{if } l = l_r \text{ for some } (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \\ \bigsqcup \{ \varphi_{l'}(\mathsf{AI}_{l'}) \mid (l', l) \in F \text{ or } (l'; l) \in F \} & \text{otherwise} \end{cases}$$

Node transfer functions (for each $l \in Lab \setminus \{ l_x \mid (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \}$):

$$\varphi_l(d) = \begin{cases} \mathsf{combine}(d, \Phi_{l_x}(\varphi_{l_c}(d))) & \text{if } l = l_r \text{ for some } (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \\ \text{specific to analysis} & \text{otherwise} \end{cases}$$

Procedure summary functions (for each $l \in Lab$ occurring in some procedure):

$$\Phi_l(d) = \begin{cases} d & \text{if } l = l_n \text{ for some } (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \\ \Phi_{l_c}(d) & \text{if } l = l_r \text{ for some } (l_c, l_n, l_x, l_r) \in \mathsf{iflow} \\ \bigsqcup \{ \varphi_{l'}(\Phi_{l'}(d)) \mid (l', l) \in F \} & \text{otherwise} \end{cases}$$

Note: introduces recursive equations for AI_l (of type D) and Φ_l (of type $D \to D$)

Dataflow Analysis for Recursive Procedures

Example 19.1 (Sign Analysis for Factorial Procedure)

Program:

```
proc [Fac(val x, res y)]^1 is
  if [x <= 0]^2 then
    [assert x <= 0]^3;
    [y := 1]^4
  else
    [assert x > 0]^5;
    [call Fac(x - 1, y)]^6_7;
    [y := y * x]^8
  end
[end]^9
```

Domain: $D = \underbrace{2^{\{+,-,0\}}}_{\text{x}} \times \underbrace{2^{\{+,-,0\}}}_{\text{y}}$ with

$\bot = (\emptyset, \emptyset)$
$\top = (\{+,-,0\}, \{+,-,0\})$

Fixpoint iteration over (Φ_8, Φ_9):
on the board

(Non-trivial) node transfer functions:

$$\begin{aligned}
\varphi_3(X,Y) &= (X \cap \{-,0\}, Y)\\
\varphi_4(X,Y) &= (X, \{+\})\\
\varphi_5(X,Y) &= (X \cap \{+\}, Y)\\
\varphi_6(X,Y) &= (\mathrm{dec}(X), \{+,-,0\})\\
\varphi_7(X,Y) &= \mathrm{combine}((X,Y), \Phi_9(\varphi_6(X,Y)))\\
\varphi_8(X,Y) &= (X, \mathrm{mlt}(Y,X))\\
\mathrm{combine}((X,Y),(X',Y')) &= (X,Y')\\
\mathrm{dec}(\{+\}) &= \{+,0\}, \ldots\\
\mathrm{mlt}(\{+,0\},\{-\}) &= \{-,0\}, \ldots
\end{aligned}$$

Procedure summary functions:

$$\begin{aligned}
\Phi_1(X,Y) &= (X,Y)\\
\Phi_2(X,Y) &= \varphi_1(\Phi_1(X,Y)) = (X,Y)\\
\Phi_3(X,Y) &= \varphi_2(\Phi_2(X,Y)) = (X,Y)\\
\Phi_4(X,Y) &= \varphi_3(\Phi_3(X,Y)) = (X \cap \{-,0\}, Y)\\
\Phi_5(X,Y) &= \varphi_2(\Phi_2(X,Y)) = (X,Y)\\
\Phi_6(X,Y) &= \varphi_5(\Phi_5(X,Y)) = (X \cap \{+\}, Y)\\
\Phi_7(X,Y) &= \Phi_6(X,Y) = (X \cap \{+\}, Y)\\
\Phi_8(X,Y) &= \varphi_7(\Phi_7(X,Y)) = \varphi_7(X \cap \{+\}, Y)\\
\Phi_9(X,Y) &= \varphi_4(\Phi_4(X,Y)) \sqcup \varphi_8(\Phi_8(X,Y))\\
&= (X \cap \{-,0\}, \{+\}) \sqcup \varphi_8(\Phi_8(X,Y))
\end{aligned}$$

Pointer Analysis

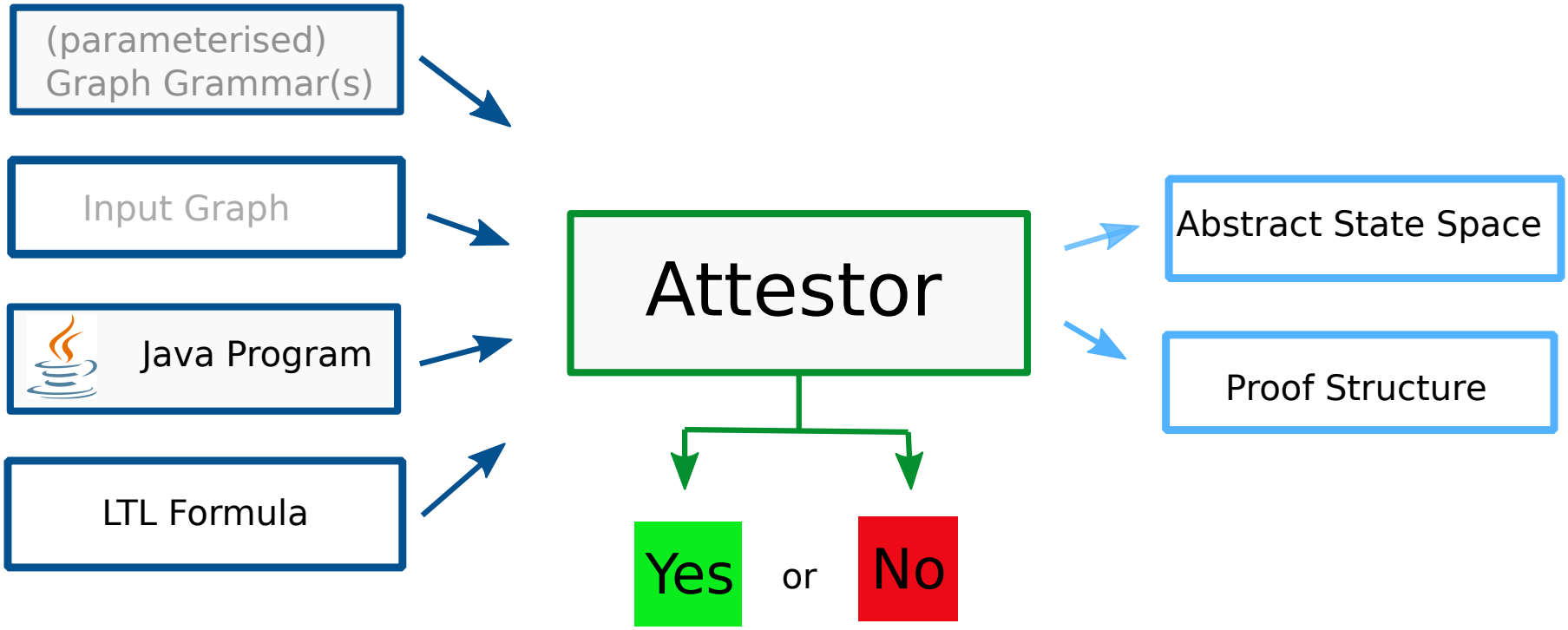
- **So far:** only static data structures (variables)
- **Now:** pointer (variables) and dynamic memory allocation using heaps
- **Problem:**
  - Programs with pointers and dynamically allocated data structures are error prone
  - Identify subtle bugs at compile time
  - Automatically prove correctness
- **Interesting properties of heap-manipulating programs:**
  - No null pointer dereference
  - No memory leaks
  - Preservation of data structures
  - Partial/total correctness

The Shape Analysis Approach

- **Goal:** determine the possible shapes of a dynamically allocated data structure at given program point
- **Interesting information:**
  - data types (to avoid type errors, such as dereferencing `null`)
  - aliasing (different pointer variables having same value)
  - sharing (different heap pointers referencing same location)
  - reachability of nodes (garbage collection)
  - disjointness of heap regions (parallelisability)
  - shapes (lists, trees, absence of cycles, ...)
- **Concrete questions:**
  - Does `x.next` point to a shared element?
  - Does a variable `p` point to an allocated element every time `p` is dereferenced?
  - Does a variable point to the head of an acyclic list?
  - Does a variable point to the root of a tree?
  - Can a loop or procedure cause a memory leak?

Pointer Analysis

The Attestor[1] Approach



[1] https://github.com/moves-rwth/attestor

Introducing Pointers

Extending the Syntax

Syntactic categories:

Category	Domain	Meta variable
Arithmetic expressions	$AExp$	a
Boolean expressions	$BExp$	b
Selector names	Sel	sel
Pointer expressions	$PExp$	p
Commands (statements)	Cmd	c

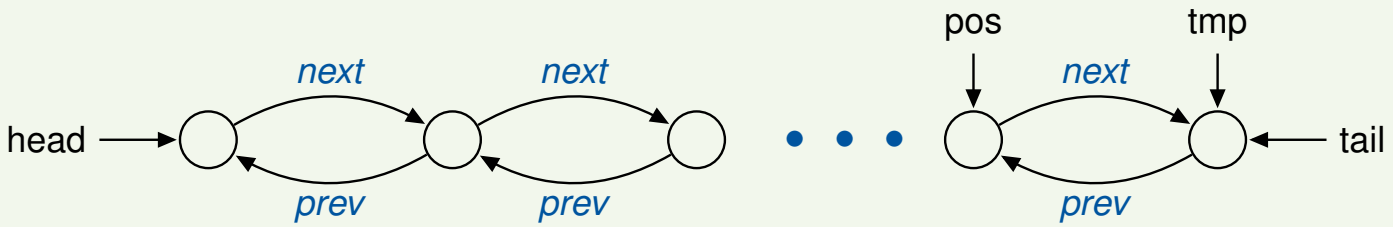
Context-free grammar

$$a ::= z \mid x \mid a_1 + a_2 \mid \ldots \mid p \mid \texttt{null} \in AExp$$
$$b ::= t \mid a_1 = a_2 \mid b_1 \wedge b_2 \mid \ldots \in BExp$$
$$p ::= x \mid x\,.\,sel$$
$$c ::= [\texttt{skip}]^l \mid [p := a]^l \mid c_1\,;\,c_2 \mid \texttt{if}\ [b]^l\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end} \mid \texttt{while}\ [b]^l\ \texttt{do}\ c\ \texttt{end} \mid [\texttt{malloc}\ p]^l \in Cmd$$

Heap Modelling & Concrete Semantics

```
pos := head;
while ¬(pos = null) do
  tmp := pos.next;
  pos.next := pos.prev;
  pos.prev := tmp;
  pos := pos.prev;
end
```

Heap representation: doubly-linked list

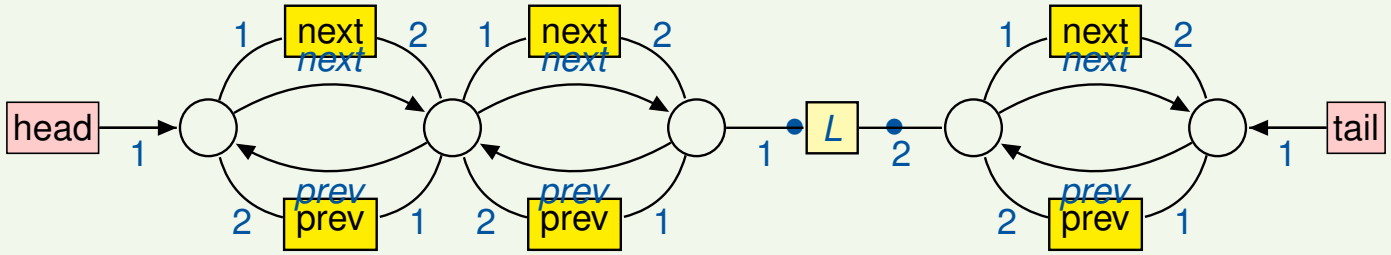



```
tmp := pos.next;
```

Hyperedge Replacement Grammars by Example

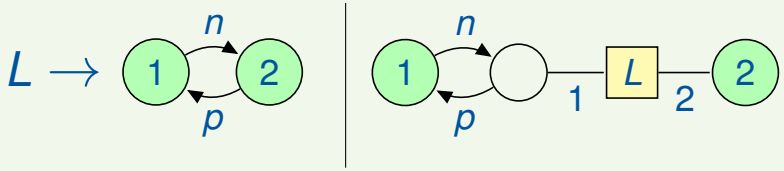
Data Abstraction

Heap representation: hypergraph



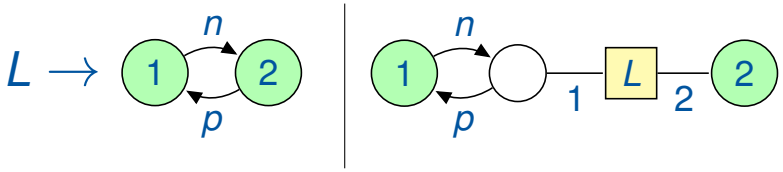
- Placeholders: nonterminal (labelled) hyperedges of rank n
- Pointers: terminal (labelled) hyperedges of rank 2
- Variables: hyperedges of rank 1

Specification of placeholder(s): Hyperedge Replacement Grammar (HRG)

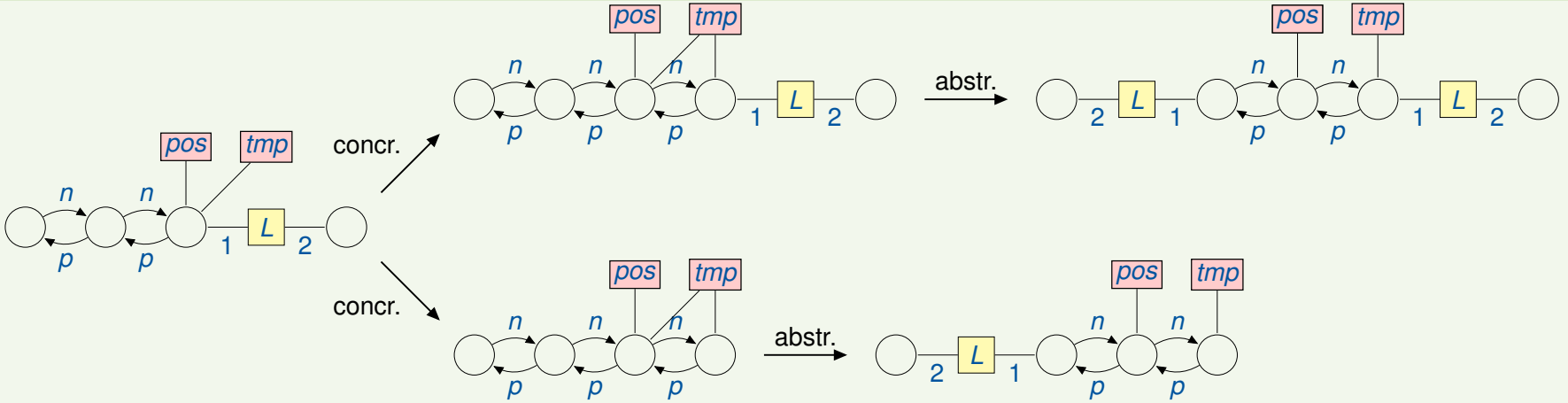


Hyperedge Replacement Grammars by Example

Abstract Execution




```
tmp := pos.next;
```



Principle

Concretise whenever necessary; abstract whenever possible.